ICST Transactions Preprint

Salus: Kernel Support for Secure Process Compartments

Raoul Strackx*, Pieter Agten*, Niels Avonds*†, Frank Piessens*

iMinds-DistriNet - KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium

Abstract

Consumer devices are increasingly being used to perform security and privacy critical tasks. The software used to perform these tasks is often vulnerable to attacks, due to bugs in the application itself or in included software libraries. Recent work proposes the isolation of security-sensitive parts of applications into protected modules, each of which can be accessed only through a predefined public interface. But most parts of an application can be considered security-sensitive at some level, and an attacker who is able to gain in-application level access may be able to abuse services from protected modules.

We propose Salus, a Linux kernel modification that provides a novel approach for partitioning processes into isolated compartments sharing the *same* address space. Salus significantly reduces the impact of insecure interfaces and vulnerable compartments by enabling compartments (1) to restrict the system calls they are allowed to perform, (2) to authenticate their callers and callees and (3) to enforce that they can only be accessed via unforgeable references. We describe the design of Salus, report on a prototype implementation and evaluate it in terms of security and performance. We show that Salus provides a significant security improvement with a low performance overhead, without relying on any non-standard hardware support.

Keywords: Privilege separation, principle of least privilege, modularization

1. Introduction

Both desktop and mobile devices are increasingly being used to perform security and privacy critical tasks, such as online banking, online tax declarations and purchasing goods from online stores. The software to perform these tasks either runs inside a web browser, or is written as a standalone application. In both cases, the software is often vulnerable to attack, either due to bugs in the application itself or due to bugs in included software libraries or in the runtime environment used to execute the application (e.g. the browser).

Because of their widespread use and potentially high-impact nature, such applications form an interesting target for cybercriminals. Past research has focused on defending against specific attack vectors such as buffer overflows [1–4], format string vulnerabilities [5] and non-control-data attacks [6]. Even though many of these defense mechanisms are applied in practice, successful attacks against high-value applications are still common.

To provide stronger security guarantees, recent research efforts have shifted from trying to defend entire applications against every possible attack to providing strong isolation of sensitive parts of an application with a minimal trusted computing base (TCB). Cryptographic keys of an application, for example, can be isolated in a protected module that has complete control over its own secrets; the module can only be accessed via its public interface. Accessing the cryptographic keys directly at assembly level is prevented by the security architecture. Thus, an attacker that has successfully exploited a vulnerability in the non-security sensitive part of the application still cannot access the cryptographic keys.

A large number of security architectures providing such protection mechanism have been proposed in this field, including software implementations using hardware virtual machine support [7, 8], trusted computing primitives [9], implementations based on system management mode [10] and even completely hardware-based solutions [11–13]. Recent research papers by Intel indicate that hardware support for these security architectures will also become available on mainstream x86 platforms in the near future [14–16].

In practice, isolating security-sensitive parts of an application is difficult since most program logic can be considered security-sensitive at some level [17]. A too

*firstname.lastname@cs.kuleuven.be
†niels.avonds@gmail.com

coarse-grained approach will result in bloated modules that may contain vulnerabilities and that are too big to be formally verified [18]. Minimum-sized modules on the other hand, can provide strong and easily verifiable guarantees, but may need to expose insecure interfaces to interact with other modules. This is a common problem of module-isolating security platforms, both in software as in hardware. Application developers are trapped in a catch-22 with possibly severe security consequences. In the recent DigiNotar attack [19], for example, the root CA's private cryptographic key was securely stored in a hardware security module (HSM), but its insecure interface enabled attackers to sign arbitrary certificates.

In order to improve upon these shortcomings, we acknowledge that almost every part of an application performs security-sensitive operations. To reduce chances of a successful attack, we propose to partition the *entire* application into compartments and implement a non-hierarchical access control mechanism between compartments. Compartments not only provide provable secure isolation of stored private data (as modules in related work do), but are also able to confine software vulnerabilities to the compartments they occur in by (1) restricting the types of system calls that they are allowed to issue, (2) enabling authentication of calling and called compartments and (3) enabling compartments to only service requests made through unforgeable references, reducing the impact of insecure interfaces. By separating likely attack vectors from attack targets and placing them into different compartments, an attacker would need to exploit vulnerabilities in multiple compartments to reach her goal.

Each compartment resides in its own chunk of memory, consisting of a *public section* containing the code of the compartment and a *private section* storing sensitive data (e.g. cryptographic keys or passwords). Only when executing the public section of a compartment can the private section of that compartment be accessed. To force other compartments to use a compartment's public interface, execution can only enter the public section via well-defined code entry points and, if required by the compartment, unforgeable references. As an additional protection measure and to support the principle of least privilege [20], compartments have the ability to restrict the types of system calls they are allowed to perform. Once a compartement drops a system call privilege, it cannot be re-acquired. This further reduces the impact of compromised compartments. The compartments of a single process all run in the same address space, providing a lightweight programming model that enables legacy applications to be ported easily and incrementally.

Consider, as an example, an X.509 certificate signing application consisting of a parser, a validator, a signer and a logging component (Figure 1). When run as a single monolithic application, a vulnerability in any one of these components can lead to the compromise of the entire application. When placing each of these components in a separate compartment under Salus, components can only call each other through their well-defined interfaces using unforgeable references and each component can authenticate both its callers and its callees. This restricts the flow of data and control between compartments to predefined patterns and raises the bar for a successful attack significantly. Consider as an example an attacker who exploited a vulnerability in the parser. In order for her to sign arbitrary certificates, she would either need to provide specially crafted credentials for the submitted certificate that would not cause the "Validator" to raise flags, or she would need to gain direct access to the "Signer" compartment by exploiting another vulnerability in the "CA Service" compartment to leak the unforgeable reference.

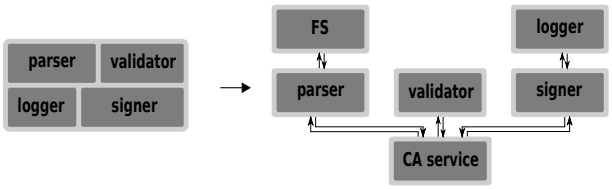


Figure 1. Salus' compartmentalization enables strong isolation of security-sensitive data *and* contains possibly vulnerable code. Multiple vulnerable compartments need to be exploited to attack the system successfully.

Furthermore, by combining unforgeable references and restricting the system calls that can be issued by a compartment, we can provide fine-grained access control to the kernel. Consider as an example the parser and assume that it reads its signing requests directly from the file system. At development time, there are two options. Option 1 is to grant the compartment access to the `open/close` and `read/write` system calls. In that case an attacker who exploited a vulnerability in the parser can inspect the entire file system with the application's privileges. The second option provides stronger security guarantees by revoking the parser compartment all system call privileges and only providing it with an unforgeable reference to a file system compartment (FS in Figure 1). This newly added compartment tightly restrict access to a single folder or file type and only provides the parser access to the files it approves. Having almost unrestricted access to the file system itself, a vulnerability in the FS compartment would enable an attacker to launch similar attacks as in option 1. However, given that this compartment is likely to be several orders of magnitude smaller than the parser compartment, the probability that such an

exploitable vulnerability can be found is limited. Such constructs are a well-known advantage of capability systems [21–23].

Concretely, we make the following contributions in this paper:

- We present a novel approach for partitioning processes into compartments with support for strong isolation of sensitive data *and* containment of vulnerabilities. To the best of our knowledge, Salus is the first solution that simultaneously (1) reduces the impact of insecure compartment interfaces, (2) enables compartments to restrict the types of system calls they are allowed to perform and (3) executes compartments in the same address space allowing legacy applications to be ported easily without having to marshall in- and output messages.
- We report on a prototype implementation of Salus in the Linux kernel.
- We evaluate the security of our approach and the performance of our prototype.

This paper is an extended version of a conference paper published at SecureComm 2013 [24]. This journal version gives a substantially extended description of the Salus system, and in addition adds support for unforgeable references to the compartment model. The remainder of this paper is structured as follows: in Section 2 we define our attacker model and describe our desired security properties. In Section 3 we provide a high-level overview of Salus, before presenting our prototype implementation in Section 4. Finally we evaluate our approach in Section 5, discuss related work in Section 6 and conclude in Section 7.

2. Attacker Model & Security Properties

We consider an attacker able to inject and execute malicious shellcode in vulnerable compartments, for example, by exploiting a buffer overflow vulnerability. Our system must defend against such attacks in the following way:

- The exploitation of a compartment must not affect the security of compartments other than those that explicitly trust the compromised compartment.
- Once a compartment is exploited, an attacker is only able to call other compartments via their proper interfaces *iff* it received a reference to those compartments. Simply guessing the compartment's virtual address is not sufficient.
- An exploited compartment may still interact with other compartments and pass compartment references. Called compartments however, will check the types of received arguments and will refuse to call other compartments with an incorrect type.
- Attackers are explicitly allowed to create new compartments. There is thus no guarantee that compartments requesting protection can be trusted. Hence, Salus must isolate compartments from one stakeholder from those of another, possibly malicious, stakeholder.
- An attacker should not be able to execute system calls that have been revoked.

Kernel-level and physical attacks are considered out of scope. Regarding the cryptographic primitives used, we assume the standard Dolev-Yao model [25]: An attacker can observe, intercept and adapt any message. Moreover, an attacker can create messages, for example by duplicating observed data. However, the cryptographic primitives used cannot be broken.

3. Overview of the Approach

This section presents a high-level overview of Salus. Section 3.1 describes the memory access control mechanisms on which Salus is based. Section 3.2 presents the services Salus provides to protected applications and section 3.3 shows how these services are used in a typical life cycle of a compartmentalized application. Authenticated communication between compartments and unforgeable references to compartments are discussed in sections 3.4 and 3.5 respectively. Finally we discuss how new and legacy applications can be compartimentalized in section 3.6.

3.1. Compartments of Least Privilege

Structure of a Compartment. The basic layout of a compartment, shown in Figure 2, is a virtual memory region divided into two sections: a public section and a private section. The *public* section contains the compartment's code and any data that should be read accessible by other compartments of the same application. This section can never be modified after initialization, which enables other compartments to authenticate the compartment based on a cryptographic hash of the public section (see Section 3.4). The start of the functions that make up the compartment's public interface are marked as entry points. Execution of the compartment can only be entered through these memory locations (see Table 1).

The *private* section contains the compartment's private data, which consists of application-specific security-sensitive data (e.g. cryptographic keys) as well as data relevant to the correct execution of the compartment, such as the runtime call stack. The data

from\to	Entry pnt.	Public section	Private section	Unprot. mem.
Entry pnt.	- - -	- - x	- - -	- - -
Public section	r - x	r - x	r w -	r w x
Private section	- - -	- - -	- - -	- - -
Unprot. mem/ other compartment	r - x	r - -	- - -	r w x

Table 1. The memory access control model enforces, for example, that a compartment's private section (4^{th} column) can only be read-write accessed from the public section of the same compartment (3^{rd} row)

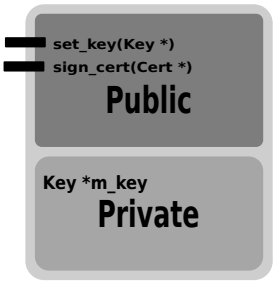


Figure 2. Salus' memory access control model enables the creation of compartments that provide strong isolation guarantees to sensitive data. Secure communication primitives reduce the impact of an insecure interface.

in the private section is read and write accessible[1] from within the compartment, but completely inaccessible for code executing outside of the compartment. Note that since each compartment has its own private call stack, intercompartmental function call arguments and return addresses must be passed via CPU registers (as opposed to passing them using the runtime stack).

Applications can still have a memory region that is not part of any compartment. This region is termed *unprotected memory* and is read/write accessible from any compartment. All compartments of the same application run in the same address space, which facilitates the compartmentalization of legacy applications. Nonetheless, fine-grained compartmentalization of a large code base can still require significant developer effort. Therefore, Salus enables applications to be compartmentalized incrementally by storing code and/or data in unprotected memory. While unprotected memory does not provide any of the security guarantees of compartments, it does provide an incremental upgrade path for legacy applications.

As an example of a compartment, consider a single compartment providing a certificate signing service. The compartment provides two functions as part of its public interface (see Figure 2). The first function, `set_key`, allows setting the cryptographic key used to sign certificates. This key is stored as the `m_key` variable in the private section. The second function, `sign_cert`, handles the actual signing requests. Salus' memory access control model ensures that only these two functions are executable; any attempt to jump to another memory location in the compartment will fail. Similarly, any attempt to directly read or write the cryptographic key in the private section from unprotected code or from another compartment will be prevented. Only after calling a valid entry point will read and write access to the private section be enabled, making the cryptographic key only accessible while the compartment is being executed. When the function is terminated, execution returns to the caller and read/write access to the compartment's private section will again be disabled.

Special care is required when execution returns to a compartment after a call to another compartment. Execution must resume at the return location, which is the instruction right after the call instruction in the caller compartment. This location however does not typically correspond to an entry point and hence would cause a memory access violation according to Salus' memory access control model (see Table 1). Compartments can implement a *return entry point* to avoid this access violation. Right before calling another compartment, the return location is placed on the top of the calling compartment's private stack while the location of the return entry point is passed to the callee in a register. When the intercompartmental call has finished, execution flow jumps to the return entry point where the return location is retrieved from the compartment's stack and jumped to. Note that a return entry point is a software implementation and follows the same access rights as any other entry point.

[1] By preventing code execution in the private section, the chances that an attacker is able to successfully exploit a vulnerability in a compartment, is reduced significantly. We acknowledge that this restriction may hinder applications that rely on generated code (e.g., JITed applications). Support for such applications could be easily added; at creation-time the creator should specify whether the new compartment's private section should be executable. As we believe this is a special case, we will not consider it for the remainder of the paper.

Restriction of Privileges. Salus provides two important primitives to limit the impact of a compromised compartment. The first primitive is caller and callee authentication. By authenticating callers and callees, a compartment can limit its interaction to trusted compartments only. Although this does not protect against trusted compartments that have been compromised, it does significantly limit the capabilities of an attacker after a successful exploit. For instance, the "signer" compartment of the CA signing service displayed in Figure 1, may only accept calls from the "CA service" compartment. As such, an attacker who successfully exploited a vulnerability in the parser may attempt to call the signing compartment, but the latter will refuse to service the attacker's service request.

The second primitive allows compartments to disable specific system calls for any code executed from within their public section. Once a system call is disabled, it cannot be re-enabled. By carefully partitioning an application into compartments, each of which should disable any system call it doesn't need, the impact of the exploitation of a vulnerable compartment is minimized. Note that much more fine-grained solutions exist than restricting complete system calls [26]. However, we focus on providing strong compartmentalization primitives that can be used as a building blocks for finer-grained privilege restriction mechanisms.

3.2. Provided Services

To enable compartmentalization of applications, Salus provides runtime support of the following services:

Create After code is loaded into memory, this service can be used to create a new compartment. Given a memory location and size for the compartment to create, Salus will enable memory protection for this region and will return a system-wide unique ID for the new compartment. Note that our attacker model explicitly allows the creation of new compartments by an attacker.

Destroy A compartment can only be destroyed by the compartment itself. After destruction, the memory access protection is disabled. Hence, a compartment should overwrite any private data before destruction.

Request compartment ID and layout To support secure communication, Salus provides a service to request the ID and layout (i.e. the size and locations of the public and private sections and the available entry points) of a compartment covering a given memory location. If there is no compartment at the specified location, the service returns an error code. This service is used as a primitive in compartment authentication.

Request caller ID To support caller authentication, Salus provides a service to request the ID and layout of the compartment that called an entry point of the current compartment.

Disable system call To limit the impact of the exploitation of a compartment, unused system calls can be disabled. To prevent an attacker from gaining system call privileges by creating a new compartment, compartments inherit system call privileges from their parent.

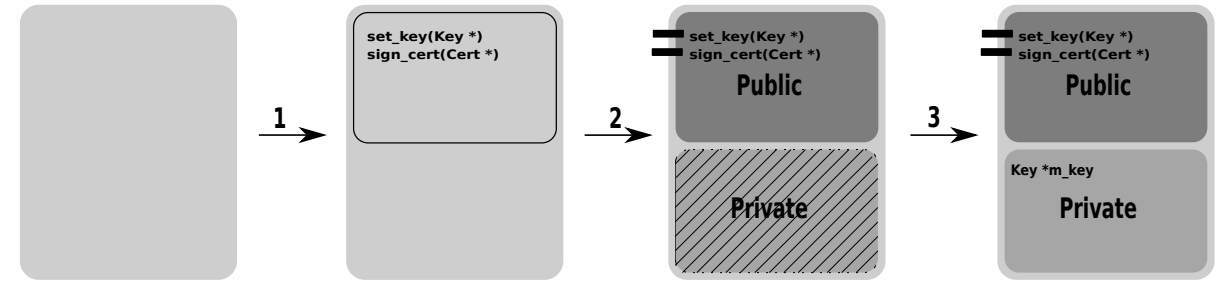
3.3. Life Cycle of a Compartmentalized Application

Compartmentalized applications can be started as any other application. After the (trusted) operating system or loader loads the application into memory and starts its execution, the application can create the required compartments. Finally, execution can jump to the compartment containing the application's main function. Compartments can be created at any point during the application's execution, for example, at the time a new (compartmentalized) plugin is loaded.

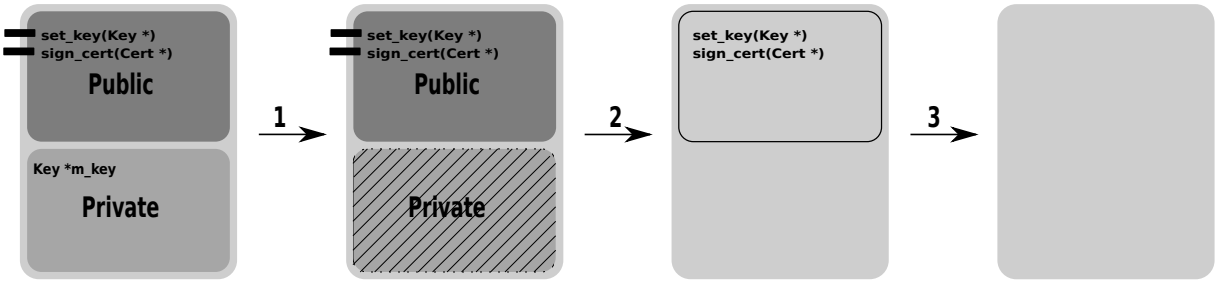
Creation of Compartments. Figure 3a shows the process of setting up a compartment. As the first step of setting up a new compartment, the application allocates (unprotected) memory and loads the compartment's code. Next, the application enables protection of this memory region, by calling Salus' creation service. Note that there is no guarantee that the new compartment's code has been loaded correctly into memory, since the creator might have been compromised already. However, any tampering with the code will be detected when the compartment tries to communicate with another compartment, as will be explained in Section 3.4.

When a new compartment is created, Salus clears the first byte of the private section. This serves as a flag to indicate to the compartment that it should initialize itself when its service is first requested. As part of its initialization, a compartment should clear the private memory locations it will use. This prevents an attacker from crafting a private section by setting it up in unprotected memory locations where a new compartment will later be created. Initialization code should typically also disable the system calls that will not be used during further execution of the compartment.

Destruction of Compartments. The destruction of a compartment, shown in Figure 3b, can only be initiated by the compartment itself. This ensures that compartments can clear their private section (which may contain sensitive data), before the memory protection is lifted. In addition, trusted communication endpoints



(a) Creation



(b) Destruction

Figure 3. The life cycle of a protected compartment from creation (3a) to destruction (3b)

could be notified of the compartment's imminent destruction. After destruction, the unprotected memory area of the destructed compartment can be freed.

3.4. Secure Communication

Salus' memory isolation mechanism provides strong guarantees that sensitive data in the private section can only be accessed by code in the public section [27–30]. Reconsidering our certificate signing service as an example (see Figure 1), we can prove that the signing key will never leave its compartment. But an attacker with access to the compartment's interface is still able to sign arbitrary certificates. Salus can limit the feasibility of such attacks in two ways: (1) by enforcing both caller and callee authentication, and (2) by requiring that callers have an unforgeable reference to the compartment at hand, which means that guessing the location of a compartment is insufficient to access it. In this section we will focus on authentication of compartments. While we will only discuss authentication of calling and called compartments, a similar approach can be applied when locations of other compartments are passed as arguments. In Section 3.5 we will show how compartments can enforce that they can only be called through unforgeable references.

Security Report. Authenticating a compartment consists of verifying whether that compartment adheres to a trusted *security report* of that compartment. A security report of a compartment consists of:

The cryptographic hash of its public section This allows any code to verify that the public section of the compartment has not been tampered with: the cryptographic hash should be recalculated at runtime and be compared to the known-good value stored in the security report. This protects against an attacker who is able to modify the public section of a compartment during its creation, before memory protection is enabled (see Section 3.3).

The layout of the compartment When a creation request originates from unprotected memory, the request itself may have been tampered with. An attacker could, for instance, specify an incorrect private section size for the compartment to create. This may result in the use of unprotected memory that should be under Salus' protection. By storing the known-good layout of the compartment in the security report, any code can verify that the layout was not tampered with during creation of the compartment.

A cryptographic signature In order to have integrity protection and authentication of the security report, it is digitally signed by its issuer. Each compartment can decide independently whether or not to trust a certain issuer, which opens up the opportunity to integrate compartments from different parties into a single application. Since the cryptographic signature provides integrity protection, security reports can simply be stored in unprotected memory.

Authentication of Called Compartments. When exchanging sensitive information between compartments, caller and callee must authenticate each other *before* sensitive data is exchanged.

To authenticate a compartment to be called, its ID must first be obtained using Salus' 'request compartment ID' service. Next, the callee's security report must be acquired. For this a central service where each compartment registers to on initialization, can be used. Given the callee's ID, the service should return the (location of the) corresponding security report. Note that this service need not be trusted, as any tampering with the information returned will be detected during the next steps. Once the security report has been obtained, it should be validated by checking the cryptographic signature and by checking that the issuer is trusted. Each compartment should contain a list of trusted security report issuers. Next, the callee compartment's layout should be requested from Salus and a hash of the public section should be calculated. The layout and the hash must be compared to the values listed in the security report. This completes the authentication and allows the caller to securely call one of the callee's public functions.

When calling a compartment that has already been authenticated in the past, a re-validation must occur because the callee may have been destroyed since the last interaction. A full authentication using the security report on every call would be very time consuming, so to reduce the performance impact, Salus allows compartments to be re-authenticated quickly based on their ID. Salus ensures each compartment has an ID that is unique on the system until the next reboot. Hence, a re-authentication can simply consist of requesting the ID of the compartment to be called (using the 'request compartment ID' service) and checking that it is the same as during the initial authentication. Using unique identifiers has the added benefit that code can distinguish between different instances of the same compartment.

Authentication of Calling Compartments. To enable compartments to limit use of their (possibly insecure) interface to trusted caller compartments, Salus provides primitives for caller authentication. For a compartment to authenticate its caller, it can first request the caller's ID and memory location (using the 'request caller ID' service) and proceed to authenticate the caller using the same steps as described above.

3.5. Unforgeable references

Salus' access control mechanism and supporting services enable authentication of both callers and callees. Unfortunately, in some situations this does not suffice. Let's reconsider the CA service from Section 1 as an example but now assume that it receives signing requests over a network. Figure 4 displays how the application can be partitioned into different compartments. A compartment `Listener` listens for incoming network connections and spawns a new `CAConnection` compartment for every connecting client. This compartment is in charge for all future communication with the client. This is similar to a `Socket` object in an object-oriented language. When a connection is established, clients must provide login credentials and a certificate request. In order to isolate vulnerabilities, `CAConnection` hands off incoming messages to a compartmentalized parser. If messages parsed correctly, the parser returns `Credentials` and `Request` compartments to the `CAConnection` compartment, or an error code if parsing failed. Once all data is collected, the `CAService` is called. Based on the provided `Credentials` and `Request` compartments, it will authenticate the client credentials, verify that the client is allowed to request a certificate for the specified domain and finally instruct the `Signer` (not displayed) to sign the certificate request.

By compartmentalizing the `Parser`, we wish to isolate possible vulnerabilities. Unfortunately, in this setup an attacker able to exploit a vulnerability in the parser may still be able to request certificates for domains that she does not own. The problem arises when the parser returns `Request` and `Credentials` compartments to `CAConnection`. Even though `CAConnection` is able to authenticate the `Parser`, it cannot verify that the received `Request` and `Credentials` compartments are based on the actual data passed to the parser. An attacker who successfully exploited a vulnerability in the parser may be able to scan[2] the entire memory and steal a `Credentials` compartment belonging to a different network connection.

To remedy the problem, we propose using unforgeable references to compartments. *Only* compartments with an unforgeable reference to a compartment have the *capability* to access it. Thus, even if a compartment was compromised, it cannot access or pass references to other compartments that it finds in memory. In our example, a compromised parser may still find a `Credentials` compartment in memory, but it is infeasible that it can guess the correct nonce (i.e., it cannot create a correct *unforgeable reference* to it). Even a compromised parser can thus not return "stolen" credentials. This results in a strict separation between different connections.

While unforgeable references in higher programming languages are easily enforceable by a type system, we

[2] An in-application level attacker may scan the entire memory in a number of ways. For example, by using Salus' service to request the layout of a compartment for likely compartment locations until a non-error result is returned, or by reading the entire program memory for telltale signs of entry points.

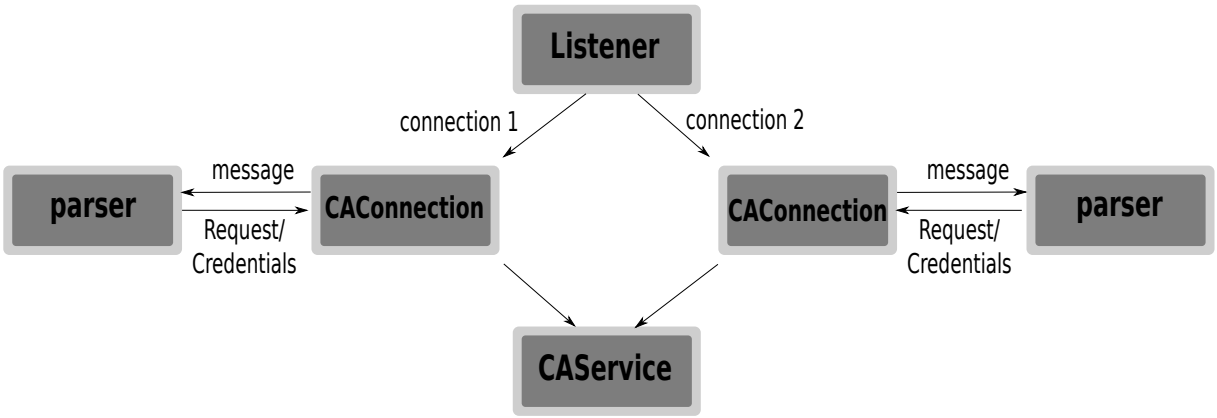


Figure 4. By enforcing that compartments can only be accessed via unforgeable references, stronger security guarantees can be guaranteed. Even if an attacker is able to exploit a vulnerability in a parser, she will be unable to access `Request/Credentials` compartments belonging to another connection.

cannot apply the same approach. An attacker able to exploit a vulnerability in a compartment has assembly-level access and can simply scan the entire memory area to access other compartments. Instead we propose establishing unforgeable references as (location, nonce) tuples. Newly created compartments must be assigned a cryptographic nonce, which can serve as a key to access the compartment's public interface. If and only if a caller provides the correct nonce, will a call to the compartment be serviced. This approach has four advantages: (1) with a sufficiently large nonce, it is computationally infeasible to forge references, (2) references can be stored in the secret section of compartments, just like any other reference, (3) compartments can implement unforgeable references using the default Salus services, and (4) both standard and unforgeable references can exist in the same application. Section 4.4 describes in detail how compartments can implement support for unforgeable references.

3.6. Writing Compartmentalized Applications

Writing safe compartments is a non-trivial task; each compartment should keep track of it's own stack, callbacks to unprotected memory should return through a specific return entry point, etc. To ease the creation of such compartments, we developed a C compiler and linker that takes care of such considerations. Application developers can simply annotate functions indicating that they are entry points, reside in unprotected memory or are located in another compartment.

Unfortunately, our tool does not solve all problems at hand. The developer is still in charge to ensure that sensitive data stored in a compartment is never accessed from unprotected memory or by another compartment. The difficulty in ensuring this depends heavily on the programming language used and the quality of the source code. Applications written in C may not be very structured. Each function may allocate memory regions and pass pointers implicit (e.g., stored in allocated memory, or type casted as an integer) or explicit (e.g., as arguments) to other functions. Compartmentalizing such legacy applications may be difficult, but given that all compartments execute in the same address space, an incremental path exists. Developers may place functions that operate on the same sensitive data in the same compartment, while initially still storing the data in unprotected memory. When all functions are placed in the compartment and sensitive data is thus only accessed by a single compartment, it can safely be allocated inside the compartment. Tools such as logging access right violations during development instead of stopping the application (as proposed by [31]) may be helpful in this process but manual inspection of code is still required.

Object-oriented languages on the other hand, may already enforce strict data encapsulation; data may only be accessed through the object's public interface. In such cases each class may be compiled as a separate compartment but to minimize overhead caused by crossing protection boundaries, multiple classes may be placed together in a single compartment.

4. Implementation

Access rights to compartment sections depend on the value of the program counter. For instance, only if execution is in the public section of a compartment, will the private section of that compartment be read/write accessible. This program counter-based memory access scheme is at the core of Salus' protection mechanism. Enforcing this scheme purely in software would have a huge performance impact as every memory access has to be checked. A pure hardware implementation of the

scheme is possible [12, 13], but prohibits its use on commodity, off-the-shelf PC platforms. The approach taken for Salus combines the best of both alternatives, by using the key insight that memory access rights for compartments only need to change when execution crosses a compartment border. This allows Salus to use the standard memory management unit (MMU) to enforce the memory protection scheme.

A prototype for Salus has been implemented as a Linux kernel modification. Section 4.1 describes how the program counter-based access control mechanism is implemented in this prototype. Section 4.2 describes the API Salus provides to processes and Section 4.3 lists the Linux system calls that had to be modified in order to provide a secure implementation of the protection mechanism.

4.1. Program Counter-Based Access Control

By aligning compartment sections to pages, the standard MMU found on any recent commodity computer can be applied to enforce the required memory protection scheme. After a compartment is created (e.g. from unprotected memory), the MMU access rights for the pages of the new compartment are set up according to Table 1: the public section is world-readable while the private section is isolated completely.

When execution tries to enter a compartment (e.g., because of a call instruction), a page fault is generated by the MMU. Based on the memory location addressed and the access type (read, write or execute), Salus determines whether a valid entry point was called and, if necessary, modifies the access rights of the calling and called compartments' public and private sections, according to Table 1. Access rights of pages unrelated to the two involved compartments are not modified, which minimizes the number of page faults and access right modifications, thereby reducing the overall performance impact.

Because unprotected memory is always readable, writable and executable, no page fault is generated when execution returns from a compartment to unprotected memory. To restore the access rights of the exited compartment, the compartment itself must issue a system call to Salus.

Since all threads of the same process normally share the same page tables, our approach cannot guarantee the required security properties in case of multiple threads. However, this is not a fundamental limitation of our model. Support for multithreaded applications can be added by modifying the kernel in order to provide each thread with a separate set of page tables. All threads have identical virtual-to-physical mappings, but with different access rights depending on the currently executing compartment in each thread. Compartments also must be multithreading-aware and provide a separate stack per thread. Our prototype currently does not support multithreading.

The Linux page fault handler was modified to implement these access right modifications. To keep track of a process' compartments, the Linux process descriptor data structure was extended with a list of `comp_struct` structures. Each `comp_struct` describes a single compartment and contains:

- The (virtual) start address and length of the public and private sections
- The compartment's unique ID
- The compartment's saved stack pointer
- A list of the compartment's remaining system call privileges

4.2. System Call API

The following new system calls were implemented in the Linux kernel. These system calls represent the API Salus provides to processes.

`void salus_create(void* start, uint len_pub, uint len_priv)` Before a new compartment is created, the list of existing compartments is checked to ensure that the new compartment will not overlap with any existing ones. New compartments must also not overlap with the kernel or have their memory pages mapped to files. When these checks succeed, a new compartment is created and added to the current process' compartment list. It receives the same system call privileges as its parent.

`void salus_destroy(void)` Since compartments can only be destroyed from within their own public section, this system call does not require any arguments. This system call restores the original memory access rights on the memory region occupied by the executing compartment and then removes the compartment from the current process' compartment list.

`struct comp_layout* salus_layout(void* addr)` This system call returns the ID and memory layout of the compartment covering a given memory location. It can be implemented by simply iterating over the current process' compartment list until a matching compartment is found. A `null` pointer is returned when there is no compartment covering the given address.

`struct comp_layout* salus_caller(void)` This system call returns the ID and memory layout of the compartment that last called an entry point

of the current compartment. A `null` pointer is returned when the current compartment was last called from unprotected memory.

`void salus_syscall_disable(uint syscall_id)` This system call disables further use of the specified system call, by removing it from the list of system call privileges in the `comp_struct` of the current compartment. Once a system call is revoked, it cannot be re-acquired.

`void salus_return(void* addr)` Before execution returns from a called compartment back to its caller (i.e. unprotected memory or another compartment), the access rights of the called compartment's pages need to be restored. This system call performs this access rights modification and then continues execution at the specified address.

4.3. Conflicting System Calls

Some existing system calls in the Linux kernel conflict with Salus' compartmentalization. Additional security checks had to be inserted for these conflicting system calls.

mprotect The `mprotect` system call can be used to change the access rights of pages in memory. Additional checks were added to prevent this system call from modifying the access rights of compartments.

mmap Existing system calls such as `mmap` or `mremap` modify the virtual address space of a process. An attacker could abuse these system calls to map a compartment's private section to a file, for instance. When the compartment then writes sensitive information to the newly mapped pages, this information may leak to an attacker. We prevent this attack by verifying that a compartment is mapped correctly before it is called. These checks were also added to the `salus_layout` API call.

personality In Linux, each process has a *personality*, which defines the process' execution domain. The personality includes, among other settings, a `READ_IMPLIES_EXEC` bit, which indicates whether read rights to a memory region should automatically imply executable rights as well. For compartments this would result in world-executable public sections, nullifying the use of designated compartment entry points. Therefore, Salus enforces that this bit is disabled for compartmentalized processes.

fork The `fork`, `vfork` and `clone` system calls can be used to create a new process or thread. As these processes or threads share parts of their page tables, the elevated access rights of the private section of a called compartment, affects all processes/threads and enable its access from unprotected memory. While these system calls could be modified to create copies of the page tables leading to the same virtual-physical address translation but with different access rights, our research prototype currently does not support this. Linux' existing `CLONE_VM` and `VM_DONTCOPY` flags are used to prevent compartments being mapped in the new process or thread. Checks were also added to the `madvice` system call, since it can be used to modify the `VM_DONTCOPY` flag.

4.4. Unforgeable references

Implementing support for unforgeable references consists of two steps: (1) newly created compartments must generate a cryptographic nonce, and (2) whenever a compartment is called, it must check whether the caller did indeed have the capability to access it.

The first step can be achieved in two ways. One option is to modify Salus' `salus_create` service call (see Section 4.2). After creating the compartment, the kernel generates a new cryptographic nonce and stores it at a specific location in the compartment's private section. Finally the `salus_create` service call returns the (location, nonce) tuple as the unforgeable reference.

Alternatively, newly created compartments can be taken ownership of on a first-call basis, by providing a `take_ownership` entry point that generates and returns an unforgeable reference on its first call. Only the first compartment that requests ownership will be provided with the unforgeable reference, subsequent calls to this entry point will be rejected. While malicous compartments may "steal" newly created compartments by taking ownership as soon as possible, they do not gain any additional power, since compartments are created from unprotected memory and hence do not possess any sensitive information that may leak to an attacker. Listing 1 shows a sample implementation of the `take_ownership` entry point in pseudo code.

In the second step, a called compartment must check whether the caller did indeed have the capability to access the compartment. To perform this check, the caller must pass the cryptographic nonce of the unforgeable reference to the called entry point. If and only if the provided nonce is identical to the nonce stored in the compartment's private section, will the call be serviced. Otherwise an error value will be returned. Note that the compartment is able to specify for every entry point whether or not it requires the nonce to

```
take_ownership:
    if ( nonce != 0 )
        return -1;
    else
    {
        nonce = gen_nonce();
        return nonce;
    }
```

Listing 1: An implementation of the `take_ownership` entry point

access it. The `take_ownership` entry point, for example, will never require a capability.

5. Evaluation

The effectiveness of Salus' protection mechanisms is evaluated in Section 5.1 and its performance impact is discussed in Section 5.2.

5.1. Security Evaluation

To evaluate Salus' security, we make a distinction between memory-safe and memory-unsafe compartments. A memory-unsafe compartment can be exploited by an attacker using low-level attack vectors such as buffer overflows [1–4], format string vulnerabilities [5] or non-control data attacks [6]. A memory-safe compartment does not contain such vulnerabilities, for instance because it was written in a memory-safe language or simply because the compartment doesn't contain any memory-safety bugs.

Since memory-safe compartments cannot be exploited directly, the only attack vector against them is through exploitation of another compartment in the same address space. However, recent research [27–30] has shown that memory protection mechanisms such as those offered by Salus, are able to provide full source code abstraction. This means that, even when other compartments have been successfully exploited, an attackers' capabilities are limited to interacting with the memory-safe compartment through its public interface. A carefully constructed interface can thus effectively limit the attack surface of a compartment. But in many cases, creating a secure interface is still a challenging problem [32]. Recall the example of a certificate signing compartment introduced in Section 3.1: even if the private cryptographic key is never exposed, an attacker could potentially still use the compartment's interface to sign arbitrary certificates [19]. By taking advantage of Salus' support for caller/callee authentication however, the risk of such an attack can be minimized by only servicing requests from compartments that would issue them as part of the normal operation of the application (e.g. in Figure 1, the signer compartment should only accept requests from the validator compartment).

Memory-unsafe compartments may still contain vulnerabilities that can be exploited by attackers. Even though Salus does not prevent such attacks, compartmentalization can still provide significant security benefits. Firstly, high-risk components can be identified and be placed in separate compartments. Effective but high-overhead countermeasures [33, 34] can be used to harden such compartments. By only applying these countermeasures to likely vulnerable compartments, their performance impact remains limited.

Secondly, Salus' ability to provide unforgeable references and it's ability to restrict access to system calls, can be used to enforce fine-grained access policies. Enabling a compartment to issue `open`/`close` and `read`/`write` system calls, essentially provides it access to the entire file system[3]. Alternatively, small, safe compartments can be created that provide similar support but may limit access to a specific folder. Since the compartment cannot issue open system calls herself, it can only access the file system through the received "capability" compartment (see Section 1 for an example).

Thirdly, compartmentalization can automatically thwart certain types of attacks. For instance, limiting entrance of compartments to valid entry points significantly reduces the chance of an attacker finding enough gadgets to successfully execute a return-oriented-programming (ROP) attack [35, 36].

Fourthly, compartmentalization can be used as a building block for new countermeasures. For instance, a custom loader could be implemented that loads compartments at different locations in memory for every program execution. This is similar to address space layout randomization (ASLR) [37], but can be applied at a much finer-grained level.

Finally, even when a compartment has been successfully exploited, Salus can still limit the impact of the attack. Because Salus provides entry point enforcement, caller/callee authentication and system call privilege containment, an attacker will likely have to compromise multiple vulnerable compartments before reaching her intended target. This significantly increases the effort an attacker must take to successfully exploit the application. The ability to confine attackers to the exploited compartment even allows implementing a tightly controlled sandbox where user-provided machine code can be executed safely.

5.2. Performance Evaluation

To evaluate the performance of Salus, we performed micro- and macrobenchmarks. All tests were run on

[3] Of course this is restricted by the access rights the application is executing in

Type	CPU cycles	Relative
Function Call	5,944	1
System Call	193,970	32.63
Compartment Call	4,024,227	677.02

Table 2. Compartment access overhead

a Dell Latitude E6510. This laptop is equipped with an Intel Core i5 560M processor running at 2.67 GHz and contains 4 GiB of RAM. A Ubuntu Server 12.04 distribution with (modified) Linux 3.6.0-rc5 x86_64 kernel was used as the operating system.

System-wide impact. To show that legacy applications not using the modularization technique are not impacted by our changes to the Linux kernel, we ran the SPECint 2006 benchmark. All tests finished within ±0.4% compared to the vanilla kernel.

Microbenchmarks. To measure the overhead caused by switching the access rights, we created a microbenchmark that measures the cost of a call to a secure compartment and compare it to the cost of calling a regular function and calling a system call. The compartment used in the benchmark immediately returns to the caller. The system call and function behave similarly.

Table 2 displays the results of this microbenchmark. Calling a compartment is about 677 times slower compared to calling a regular function. This overhead is attributed to the need to modify the access rights of pages. Compared to calling a system call, the compartment is only 20 times slower. Due to these high costs, there is a trade-off to be made between a low number of compartment transitions and small compartments with additional security guarantees.

Secure Web Server. As a macrobenchmark, we compartmentalized an SSL-enabled web server based on an example provided by PolarSSL library[4]. For every new connection a new compartment is created, securing session keys even in the event that an attacker is able to inject shellcode in the compartment providing its own SSL session.

The secure compartment was built using the PolarSSL cryptographic library and a subset of the diet libc library. A simple static 74-byte page is returned to the clients over an SSL-connection protected by a 1024-bit RSA encryption key.

We used the Apache Benchmark to benchmark this web server for an increasing number of clients that are concurrently requesting pages. The results are shown in Table 3. The performance overhead tops at 12.72%

[4] https://polarssl.org/

Concurrency	Vanilla	Salus	Relative perf.
1	109.11	96.54	-11.52 %
2	165.56	153.62	-7.21 %
4	184.31	164.78	-10.60 %
8	199.98	175.35	-12.32 %
16	206.82	181.00	-12.48 %
32	207.78	181.50	-12.65 %
64	206.64	180.35	-12.72 %
128	206.49	180.97	-12.36 %

Table 3. Requests per second of an SSL-enabled webserver where every SSL session is protected in its own compartment, for an increasing number of clients.

and is mainly attributed to the many compartment boundaries crosses during the SSL negotiation phase.

Compartmentalized parser. As input files are often under the control of an attacker and sanitation of their content can be difficult, parsers are a likely attack vector for many applications. As a second benchmark, we isolated the decompressing function of gzip (GNU zip). While disabling unused system calls for the entire process would result in similar security guarantees, we are interested in the impact of repeated compartment crossings in a parser setting. Applications that place their parser and the rest of the application in different compartments, would incur a similar overhead as only one additional compartment boundary needs to be crossed.

To benchmark the application, we created input files with randomized content, ranging from 16 KiB to 64 MiB in size, compressed them and measured the time taken to decompress the files with the hardened application. The application was run 100 times on each file. File I/O used a buffer of 32 KiB and the output was redirected to the null device. Figure 5 displays the results.

Given the relatively high overhead of a call to a compartment and the low computation cost of the decompressing function, it is unsurprising that for small input files the overhead can be as high as 21.9%. When the input size is increased however, the overhead drops steadily to -0.5% for 64 MiB input files, even though also the number of compartment-border crossings increases from 8 to 8200. We attribute this significant drop in overhead to the increased amount of slow disk I/O that needs to be performed as the input file size gets bigger, an effect that we predict to see in most parser-like compartments. The small performance gain of 0.5% can be attributed to cache effects.

The way an application is partitioned will have a significant impact on performance. Applications should be compartmentalized in logical blocks where each compartment has direct access to most of its required

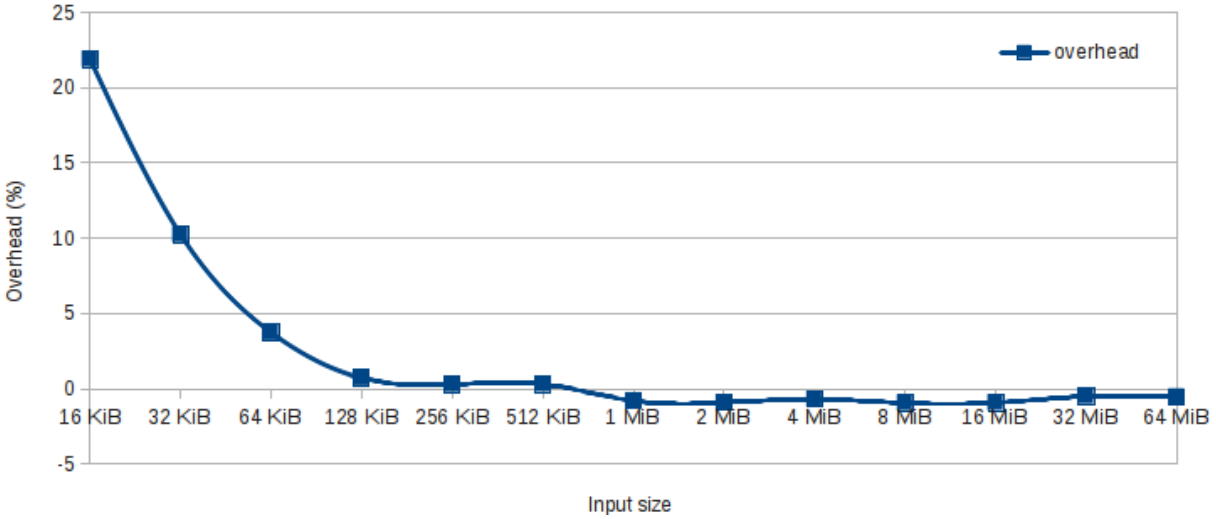


Figure 5. Salus' performance overhead on the gzip macro benchmark drops significantly as the input file size increases.

data. Once a logical block has finished, control and all data should be passed to the next compartment, reducing the number of inter-compartment calls. Smaller, heavily protected compartments such as an SSL compartment, provide strong security but may impact performance more significantly when called repeatedly. This makes the performance impact of compartmentalization difficult to predict. Therefore we advocate for automatic partitioning tools that analyze the application's call graph and information flow to reduce the number of compartment crosses and help the programmer decide which compartments should be hardened.

6. Related Work

Various security measures have been proposed to harden applications. Many of them aim to protect against very specific vulnerabilities such as buffer overflows [1–4], format string vulnerabilities [5] or non-control data attacks [6]. While these countermeasures make it significantly more difficult for an attacker to compromise software applications, they cannot offer complete protection. Static verification of source code [38], in contrast, is able to provide such hard security guarantees, but typically comes at a significant economic cost in terms of programming and verification effort.

Singaravelu et al. [17] proposed to isolate security-sensitive parts of applications in complete isolation from the rest of the system. Many research proposals have since been filed based on this principle. Each of them provides some way of executing modules in isolation, relying on a trusted code base ranging from only a few thousands of lines of code [7, 8] to only the protected modules themselves and a small runtime library [9, 10]. More recently, specially tailored hardware support has been proposed in academia [11–13] and industry [14–16]. While these research prototypes offer provable security to the sensitive data that they protect [27–30], they do not attempt to reduce the impact of a vulnerability elsewhere in the code by executing modules with the least amount of privileges possible [20]. An attacker who successfully gains control over the platform is still able to interact with other protected modules unrestrictedly.

Other work focuses on confining possible software vulnerabilities. Early work focused on reducing the size of the kernel itself [39], where process privileges are managed by capabilities. Recently Watson et al. [26] proposed applying a similar idea to partition applications themselves, where capabilities can be granted to each created partition. As partitions live in their own process, interaction takes place through remote procedure calls and passed data must be marshalled. Salus avoids these drawbacks by executing compartments in the same address space and unprotected memory can be used to gradually partition legacy applications (see section 3.6).

Provos et al. [40] and Brumley et al. [41] propose separating sensitive applications into a privileged monitor and one or multiple slave components. Monitor and slaves communicate through system sockets and thus also require arguments to be marshalled. Subsequent work by Provos [42] argues for finer grained access policies for system calls. Bittau et al. [31] also propose splitting applications into compartments (called sthreads) executing with least privilege. Developers can tag memory locations and a security policy enforces that a compartment can

only access memory locations with a matching tag. When an sthread requires more privilege operations, it can request so by calling a callgate. A security policy enforces which callgates an sthread can call. Salus' unforgeable references enable a much more flexible security policy. Compartments can be provided temporary access to system resources by encapsulating them in a compartment. As all interaction to the resource passes by this compartment, the caller's access rights can easily be revoked at a later point in time [43].

Native Client (NaCl) [44, 45], which builds upon the concepts of software fault isolation [46], takes another approach and attempts to completely sandbox x86 code. Accesses to the environment from within a sandbox are tightly controlled by runtime facilities. While NaCl focuses on downloaded, untrusted binary code, it could be used to partition entire applications. Interaction between two NaCl partitions is provided through a service similar to Unix domain sockets, making porting existing legacy applications a challenging undertaking. Salus on the other hand can provide a similar tightly controlled sandbox by placing such partitions in one compartment while the remaining legacy application is placed in another. A specially created wrapper can ensure that all system call privileges are revoked before execution control is given to the sandboxed code. There are however two major differences compared to NaCl. First, Salus only impacts performance when compartment boundaries are crossed. NaCl on the other hand places constraints on the binary code itself, resulting in a varying performance impact. Second, Salus employs a non-hierarchical separation of privilege, allowing compartments to be completely isolated from other compartments (possibly provided by other vendors) while compartments of the same vendor can co-operate easily.

Finally, our earlier work [8, 11] is the most related to Salus. It also employs a program-counter based access control mechanism, but assumes a safe interface. Therefore it has the same limitation as other research prototypes [7, 9, 10] that provide strong isolation of sensitive data: it does not reduce the possible impact of exploited vulnerabilities.

7. Conclusion

Protected-module architectures isolate sensitive parts of applications. They guarantee that sensitive data can only be accessed via a well-defined interface. In practice, however, it is hard to isolate security-sensitive parts, as most code in an application is sensitive up to some level. As a result, modules of such platforms may need to provide unsafe interfaces; an attacker may not access the sensitive data directly, but access to the provided interface may still lead to unwanted behavior.

We presented Salus, a new security architecture providing strong isolation guarantees of both sensitive data and software vulnerabilities. Salus significantly reduces the impact of unsafe interfaces by (1) supporting the authentication of compartments and (2) enabling compartments to enforce that they can only be accessed through unforgeable references. This allows likely attack vectors and targets to be placed in different compartments, such that an attacker must successfully attack multiple compartments before an attack target can be reached.

Acknowledgement. This work has been supported in part by the Intel Lab's University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). Raoul Strackx holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT). Pieter Agten holds a PhD fellowship of the Research Foundation - Flanders (FWO).

References

[1] Aleph One: Smashing the stack for fun and profit. Phrack magazine **7**(49) (1996)

[2] Erlingsson, Ú.: Low-level software security: Attacks and defenses. In Aldini, A., Gorrieri, R., eds.: Foundations of Security Analysis and Design IV. Volume 4677 of Lecture Notes in Computer Science. Springer-Verlag (2007) 92–134

[3] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: Proceedings of the Second European Workshop on System Security. EuroSec'09, New York, NY, USA, ACM (2009) 1–8

[4] Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Department of Computer Science, KULeuven (2004)

[5] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: Formatguard: automatic protection from printf format string vulnerabilities. In: Proceedings of the 10th conference on USENIX Security Symposium. SSYS'01, Berkeley, CA, USA, USENIX Association (2001) 1–9

[6] Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the 14th conference on USENIX Security Symposium. Volume 14 of SSYM'05., Berkeley, CA, USA, USENIX Association (2005) 177–192

[7] McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB reduction and attestation. In: Proceedings of the IEEE Symposium on Security and Privacy. S&P'10, Washington, DC, USA, IEEE Computer Society (May 2010) 143–158

[8] Strackx, R., Piessens, F.: Fides: Selectively hardening software application components against kernel-level or process-level malware. In: Proceedings of the 19th ACM

conference on Computer and Communications Security. CCS'12, New York, NY, USA, ACM (October 2012) 2–13

[9] McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: Proceedings of the ACM European Conference in Computer Systems. EuroSys'08, New York, NY, USA, ACM (April 2008) 315–328

[10] Azab, A., Ning, P., Zhang, X.: SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In: Proceedings of the 18th ACM conference on Computer and communications security. CCS'11, New York, NY, USA, ACM (2011) 375–388

[11] Strackx, R., Piessens, F., Preneel, B.: Efficient Isolation of Trusted Subsystems in Embedded Systems. In Jajodia, S., Zhou, J., eds.: Security and Privacy in Communication Networks (SecureComm'10). Volume 50 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering., Springer Berlin Heidelberg (2010) 344–361

[12] Noorman, J., Agten, P., Daniels, W., Strackx, R., Herrewege, A.V., Huygens, C., Preneel, B., Verbauwhede, I., Piessens, F.: Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In: 22nd USENIX Security Symposium. SSYM'13, USENIX Association (August 2013)

[13] Owusu, E., Guajardo, J., McCune, J., Newsome, J., Perrig, A., Vasudevan, A.: OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. CCS'13, New York, NY, USA, ACM (2013) 13–24

[14] Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. Volume 13 of HASP'13., New York, NY, USA, ACM (2013)

[15] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., Del Cuvillo, J.: Using innovative instructions to create trustworthy software solutions. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. HASP'13, New York, NY, USA, ACM (2013) 11

[16] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. HASP'13, New York, NY, USA, ACM (2013) 8

[17] Singaravelu, L., Pu, C., Härtig, H., Helmuth, C.: Reducing TCB complexity for security-sensitive applications: three case studies. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems. EuroSys'06, New York, NY, USA, ACM (2006) 161–174

[18] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: Operating Systems Review. Volume 37 of OSR'03., New York, NY, USA, ACM (2003) 193–206

[19] Hoogstraten, H., Prins, R., Niggebrugge, D., Heppener, D., Groenewegen, F., Wettinck, J., Strooy, K., Arends, P., Pols, P., Kouprie, R., Moorrees, S., van Pelt, X., Hu, Y.Z.: Black Tulip - report of the investigation into the DigiNotar certificate authority breach. Technical report, FoxIT (2012)

[20] Saltzer, J., Schroeder, M.: The protection of information in computer systems. In: Proceedings of the IEEE. Volume 63., IEEE (1975) 1278–1308

[21] Carter, N.P., Keckler, S.W., Dally, W.J.: Hardware support for fast capability-based addressing. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS'94, New York, NY, USA, ACM (1994) 319–327

[22] Woodruff, J., Watson, R.N.M., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R., Roe, M.: The CHERI capability model: Revisiting RISC in an age of risk. In: Proceedings of the 41st International Symposium on Computer Architecture. ISCA'14 (2014)

[23] Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Communications of the ACM **9** (March 1966) 143–155

[24] Avonds, N., Strackx, R., Agten, P., Piessens, F.: Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In Zia, T., Zomaya, A., Varadharajan, V., Mao, M., eds.: Security and Privacy in Communication Networks (SecureComm'13). Volume 127 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering., Springer International Publishing (September 2013) 252–269

[25] Dolev, D., Yao, A.C.: On the security of public key protocols. In: IEEE Transactions on Information Theory. Volume 29., Piscataway, NJ, USA, IEEE Press (September 1983) 198–208

[26] Watson, R.N., Anderson, J., Laurie, B., Kennaway, K.: Capsicum: practical capabilities for UNIX. In: Proceedings of the 19th USENIX Security symposium. SSYM'10, Berkeley, CA, USA, USENIX Association (2010)

[27] Agten, P., Strackx, R., Jacobs, B., Piessens, F.: Secure compilation to modern processors. In: Proceedings of the 25th Computer Security Foundations Symposium. CSF'12, Los Alamitos, CA, USA, IEEE Computer Society (2012) 171–185

[28] Patrignani, M., Clarke, D.: Fully abstract trace semantics of low-level isolation mechanisms. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. SAC'14, ACM (March 2014) 1562–1569

[29] Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., Piessens, F.: Secure compilation to protected module architectures. In: Accepted for publication in Transactions on Programming Languages and Systems (TOPLAS), New York, NY, USA, ACM

[30] Patrignani, M., Clarke, D., Piessens, F.: Secure Compilation of Object-Oriented Components to Protected Module Architectures. In Shan, C.c., ed.: Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13). Volume 8301 of Lecture Notes

in Computer Science., Springer International Publishing (2013) 176–191
[31] Bittau, A., Marchenko, P., Handley, M., Karp, B.: Wedge: Splitting applications into reduced-privilege compartments. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. NSDI'08, Berkeley, CA, USA, USENIX Association (2008) 309–322
[32] Longley, D., Rigby, S.: An automatic search for security flaws in key management schemes. Computers & Security **11**(1) (1992) 75–89
[33] Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: Paricheck: an efficient pointer arithmetic checker for c programs. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security. ASIACCS '10, New York, NY, USA, ACM (2010) 145–156
[34] Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th conference on USENIX security symposium. SSYM'09, USENIX Association (2009) 51–66
[35] Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM conference on Computer and communications security. CCS '07, New York, NY, USA, ACM (2007) 552–561
[36] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS'10, New York, NY, USA, ACM (2010) 559–572
[37] Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX security symposium. Volume 12 of SSYM'03., Berkeley, CA, USA, USENIX Association (2003) 105–120
[38] Jacobs, B., Piessens, F.: The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven (August 2008)
[39] Liedtke, J.: Toward Real Microkernels. Communications of the ACM **39**(9) (1996) 77
[40] Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: Proceedings of the 12th Conference on USENIX Security Symposium. SSYM'03, Berkeley, CA, USA, USENIX Association (2003)
[41] Brumley, D., Song, D.: Privtrans: Automatically partitioning programs for privilege separation. In: Proceedings of the 13th Conference on USENIX Security Symposium. Volume 13 of SSYM'04., Berkeley, CA, USA, USENIX Association (2004)
[42] Provos, N.: Improving host security with system call policies. In: Proceedings of the 12th Conference on USENIX Security Symposium. SSYM'03, Berkeley, CA, USA, USENIX Association (2003)
[43] Miller, M., Yee, K.P., Shapiro, J.S.: Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University (2003)
[44] Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: A sandbox for portable, untrusted x86 native code. In: Proceedings of the 30 IEEE Symposium on Security and Privacy. S&P'09, IEEE (2009) 79–93
[45] Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., Chen, B.: Adapting software fault isolation to contemporary CPU architectures. In: Proceedings of the 19th USENIX Security Symposium. SEC'10 (2010)
[46] Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: Proceedings of the fourteenth ACM symposium on Operating systems principles. SOSP '93, New York, NY, USA, ACM (1993) 203–216